ARIEL UNIVERSITY

DEEP LEARNING FINAL PROJECT

Sudoku solver

Author:
Almog Nataly & Elmalech Ido

Department of Computer Science

December 28, 2021

Ariel University

Abstract

Faculty of Natural Sciences
Department of Computer Science

Sudoku solver

by Almog Nataly & Elmalech Ido

Sudoku is a very popular mind-thinking game. Recent advances in the world of computer science provide us with machine learning algorithms. In this paper, we propose a Convolutional Neural Network (CNN) model suitable for solving a Soduko game. We propose several modifications to the existing learning algorithm to make it more suitable under the game rules, namely 1. We look at the game in segments of 9. 2. We employ "out of the box" thinking to solve the game in a more human approach.

Contents

Chapter 1

Introduction

Sudoku is a popular number puzzle that requires you to fill blanks in a 9X9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids contains all of the digits from 1 to 9. There have been various approaches to solving that, including computational ones. In this project, We show that simple convolutional neural networks have the potential to crack Sudoku without any rule-based postprocessing.

In this paper, we will build a machine learning model that will be able to crack a Sudoku game. Using a neural network we will try to predict the result of a Sudoku game in the following method: A straight forward approach to solve the game. Trying to predict all the missing numbers and get a complete solution in one guess.

We tried a few different network architecture and strategies to test which one will give us the best results with the dataset which we have been given.

Chapter 2

Project description

2.1 Experimental data

2.1.1 Data Collection

We found the following data on Kaggle, which contains 1 million unsolved and solved Sudoku games.
The dataset was received as a CSV file and contains 2 columns. The column *quizzes* has the unsolved games and represent the features, the column *solutions* has respective solved games and represent the labels. Each game is represented by a string of 81 numbers as follow:

	A	B
1	quizz	solution
2	'004300209005009001070060043006002087190007400050083000600000105003508690042910300	'864371259325849761971265843436192587198657432257483916689734125713528694542916378
3	'040100050107003960520008000000000017000906800803050620090060543600080700250097100	'346179258187523964529648371965832417472916835813754629798261543631485792254397186
4	'600120384008459072000006005000264030070080006940003000310000050089700000502000190	'695127384138459672724836915851264739273981546946573821317692458489715263562348197
5	'497200000100400005000016098620300040300900000001072600002005870000600004530097061	'497258316186439725253716498629381547375964182841572639962145873718623954534897261
6	'005910308009403060027500100030000201000820007006007004000080000640150700890000420	'465912378189473562327568149738645291954821637216397854573284916642159783891736425
7	'100005007380900006000004808200010750407600200690020010050390040000201000000046352	'194685237382974516657213489823491675541768923769352841215839764436527198978146352

FIGURE 2.1: Sudoku Dataset

2.1.2 Data Processing

Our task is to feed the unsolved sudoku to a neural network and get the solved sudoku out of it. This means we have to feed 81 numbers to the network and need to have 81 output numbers from them.

By using NUMPY we converted the input data (unsolved games) into a 3D array of shape (9,9,1) to represent a 9x9 sudoku converted from the string. The number 0 represents a blank position in the unsolved game.

We will be normalizing our data between 0 and 1 by dividing all the numbers by 9 and then subtracting 0.5 from it.

Example:

Input (Unsolved):

$$\begin{bmatrix} 0 & 1 & 6 & 9 & 0 & 4 & 0 & 0 & 7 \\ 0 & 0 & 4 & 0 & 3 & 0 & 0 & 8 & 0 \\ 0 & 0 & 3 & 0 & 6 & 1 & 9 & 2 & 0 \\ 5 & 0 & 9 & 1 & 4 & 0 & 8 & 0 & 0 \\ 1 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 7 & 0 & 0 & 0 & 6 & 5 \\ 6 & 0 & 0 & 0 & 0 & 2 & 0 & 4 & 0 \\ 0 & 2 & 0 & 8 & 0 & 5 & 3 & 1 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 9 \end{bmatrix}$$

Output (Solved):

$$\begin{bmatrix} 2 & 1 & 6 & 9 & 8 & 4 & 5 & 3 & 7 \\ 2 & 9 & 4 & 2 & 3 & 7 & 6 & 8 & 1 \\ 7 & 8 & 3 & 5 & 6 & 1 & 9 & 2 & 4 \\ 5 & 6 & 9 & 1 & 4 & 6 & 8 & 7 & 3 \\ 1 & 7 & 2 & 5 & 5 & 8 & 4 & 9 & 4 \\ 4 & 4 & 8 & 7 & 2 & 9 & 1 & 6 & 5 \\ 6 & 9 & 1 & 3 & 1 & 2 & 5 & 4 & 8 \\ 9 & 2 & 7 & 8 & 9 & 5 & 3 & 1 & 6 \\ 4 & 3 & 1 & 4 & 1 & 6 & 2 & 5 & 9 \end{bmatrix}$$

2.2 Architecture

- Input: batch of vectors of shape (9, 9)
- Output: vector of shape (1, 81)
- Batch: our batch size 32.
- Data split: data is split to 60% and 40%
- We will be using reducing lr approach to finetune our model.
- Checkpointing to save the best model and avoid overfitting.

Note: fully connected has 729 neurons to stimulate having 81 * 9 combinations for a fully solved 9x9 square.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 9, 9, 32)          320
_________________________________________________________________
batch_normalization (BatchNo (None, 9, 9, 32)          128
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 9, 9, 64)          18496
_________________________________________________________________
batch_normalization_1 (Batch (None, 9, 9, 64)          256
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 128)         8320
_________________________________________________________________
flatten (Flatten)            (None, 10368)             0
_________________________________________________________________
dense (Dense)                (None, 729)               7559001
_________________________________________________________________
reshape (Reshape)            (None, 81, 9)             0
_________________________________________________________________
activation (Activation)      (None, 81, 9)             0
=================================================================
Total params: 7,586,521
Trainable params: 7,586,329
Non-trainable params: 192
```

2.3 Training

We trained the network for 25 epochs, with batch size 32. The learning rate started at 0.001 and decreased exponentially after 10 epochs. We have realized that leaving the learning rate at 0.001 is better for us. The final training loss settled down to 0.3.. We tried a few different network architecture and strategies but could not reduce the loss further so we went ahead with this network. The accuracy of the network settled down at about 0.85, every attempt to push the accuracy beyond did not work.

comparisons with different hyper-parameters can be shown in chapter 4.

Chapter 3

Simulation results

After training our model for a few days and over 25 epochs, we have come to realize we can't decrease our test loss beyond 0.3x percent, and accuracy above 0.8x percent. In reality, running about 2-3 epochs is as equal to running 25 epochs.

Below, showing comparison results for various hyperparameters:

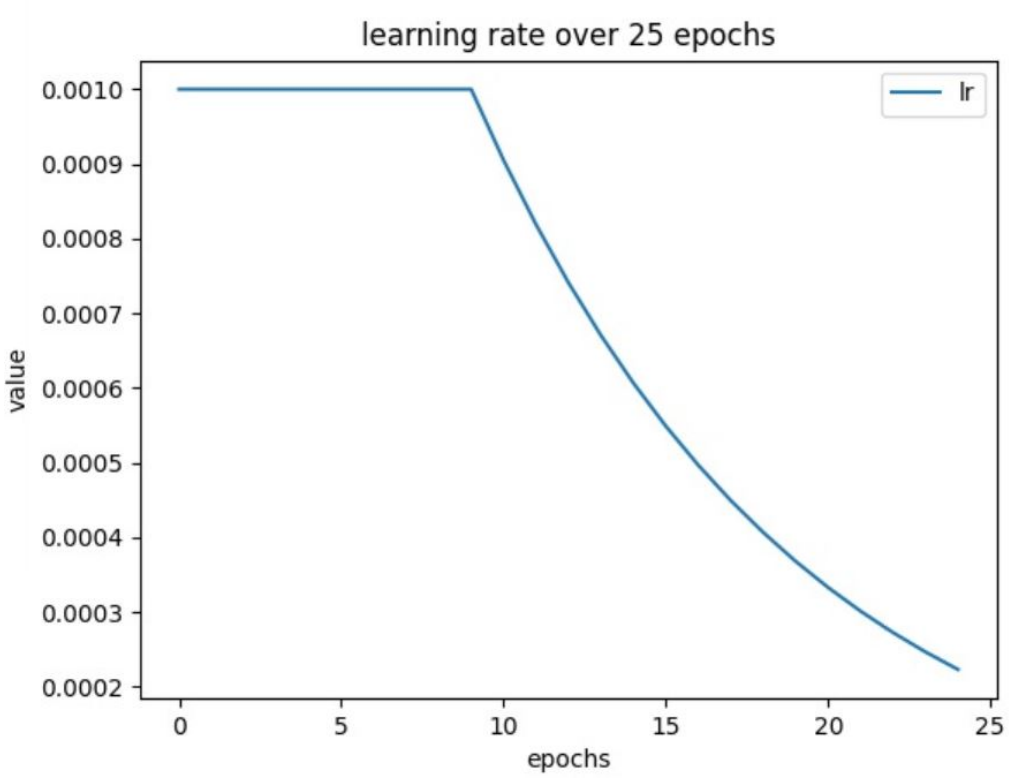

FIGURE 3.1: **Learning rate.** The initial learning rate for the first 10 epochs is 0.001, and after 10 epochs it is decreased exponentially by the following formula: $lr \cdot e^{-0.1}$

3.1 Optimizers for each batch size 32 / 64 / 128

To try achieving a better model, we have tested a few optimizers: Adam, SGD, RMS. We will show comparisons by batch size.

3.1.1 batch size 32

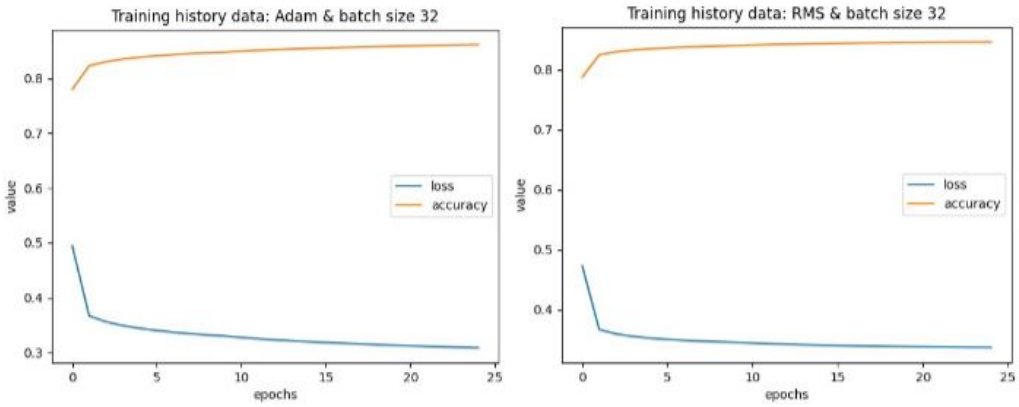

FIGURE 3.2: Test loss and accuracy by epochs. As we can see Adam and RMS produce almost exactly the same results.

FIGURE 3.3: Test loss and accuracy by epochs. SGD results are very un-accurate than Adam or RMS so we decided to give up on it.

3.1.2 batch size 64

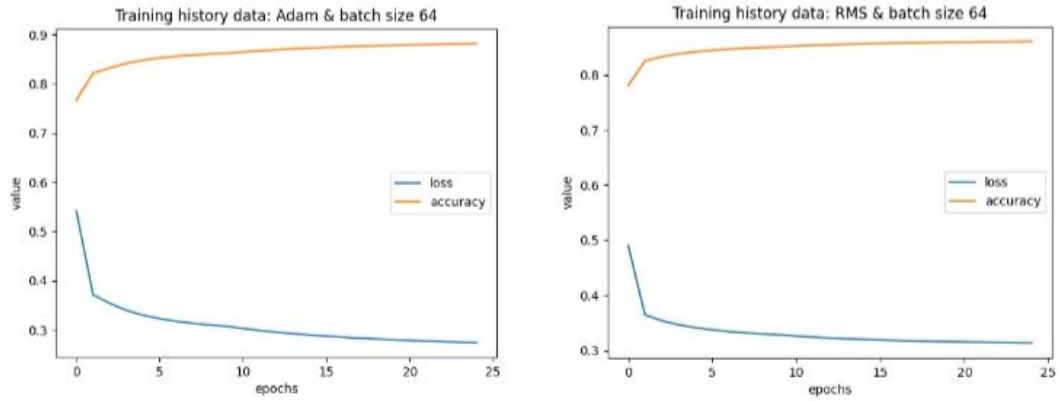

FIGURE 3.4: Same results for batch size 64. Adam did a bit better reaching almost 0.9.

3.1.3 batch size 128

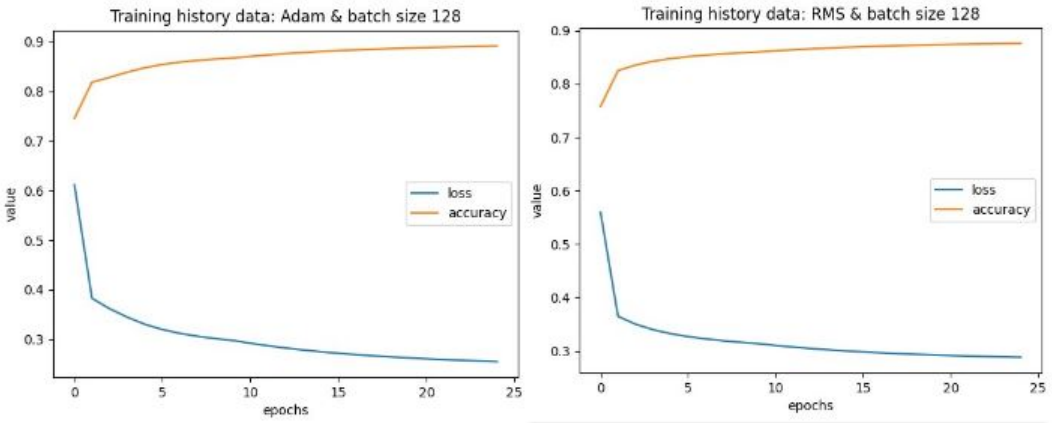

FIGURE 3.5: Same results for batch size 128. Adam and RMS did almost exactly the same, both reaching 0.9 percent.

3.2 Final results

3.2.1 accurancy

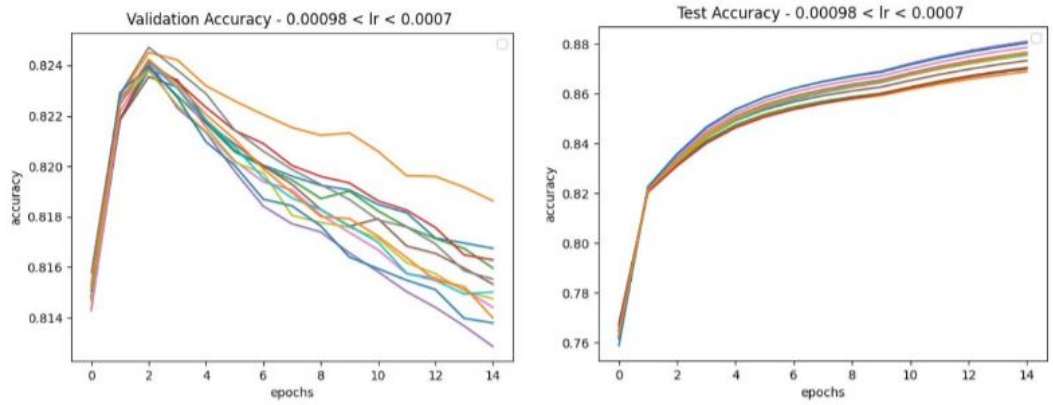

FIGURE 3.6: Test and Validation accuracy by LR tuning.

3.2.2 loss

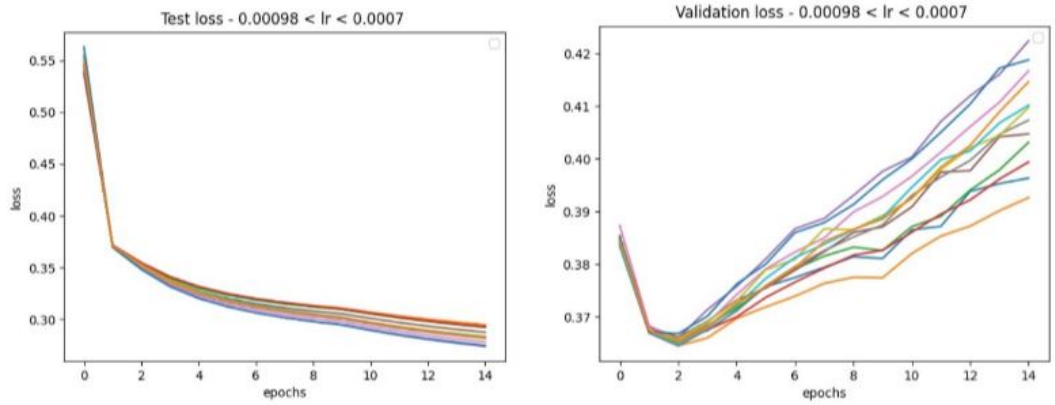

FIGURE 3.7: Test and Validation loss by LR tuning.

3.2.3 Final model

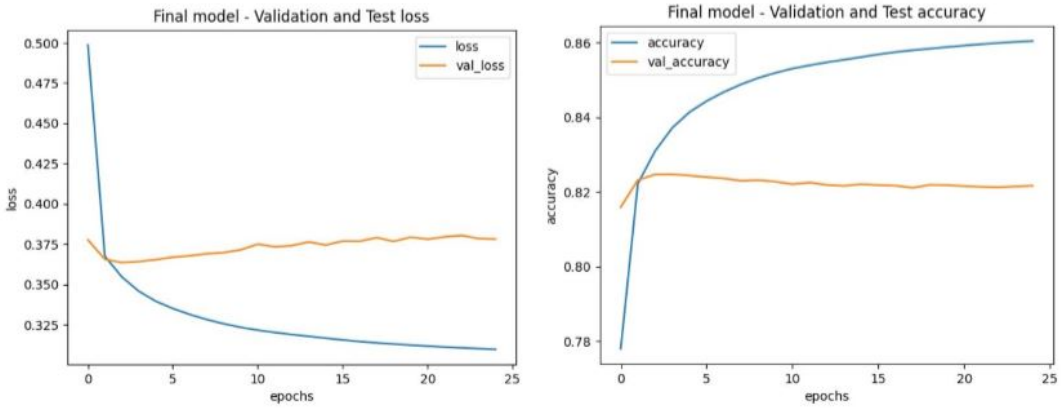

FIGURE 3.8: Final model results: Adam, batch size 32, lr .001

Chapter 4

Conclusions

To conclude, our network has reached 85% accuracy for our data. We have found over-fitting while trying different models, it seems the network can't find any path for solving a complete sudoku game at once. One might think 85% for solving a Sudoku is very accurate. In reality our machine achieved 85% accuracy for the data, meaning there is 85% accuracy for the solved game to be similar to its actual solved game. Running a prediction over a game does not look that solved after all, meaning our machine fails almost every time at solving a game.

We asked a simple quest: What can be done dramatically to increase the accuracy of our machine? The answer is to think like a human – solve the game number by number.

Achieving this can be done in two ways:

1. After training our model, we can use it to predict a single digit to be solved. After that, feed the network the same sample with the solved digit inside. that way our machine can get a "feeling" of solving the game one digit after another.
   Disadvantages: the network doesn't actually learn to play single digits.

2. Change the network entirely to a different approach – using Reinforcement learning with a DQN network and an Agent who can fill in a digit one by one while training.
   Disadvantages: the network does not compare to a solved solution using RL, a reward function has to be written.

Utilizing suggested solution 1, we have built a function that uses a single digit from the machine's prediction and fill it in the sample just as described, returning the solved Sudoku game after all digits are filled in.

The results were really amazing! We have managed to "increase" the accuracy of the machine to about 99.9%!

Running a validation on 1 million games is quite ambitious, results are shown in a table below for 1000 games:

games	model	one by one
10	0	10
50	0	50
100	0	100
500	0	500
1000	0	999